COMP 250

Lecture 22

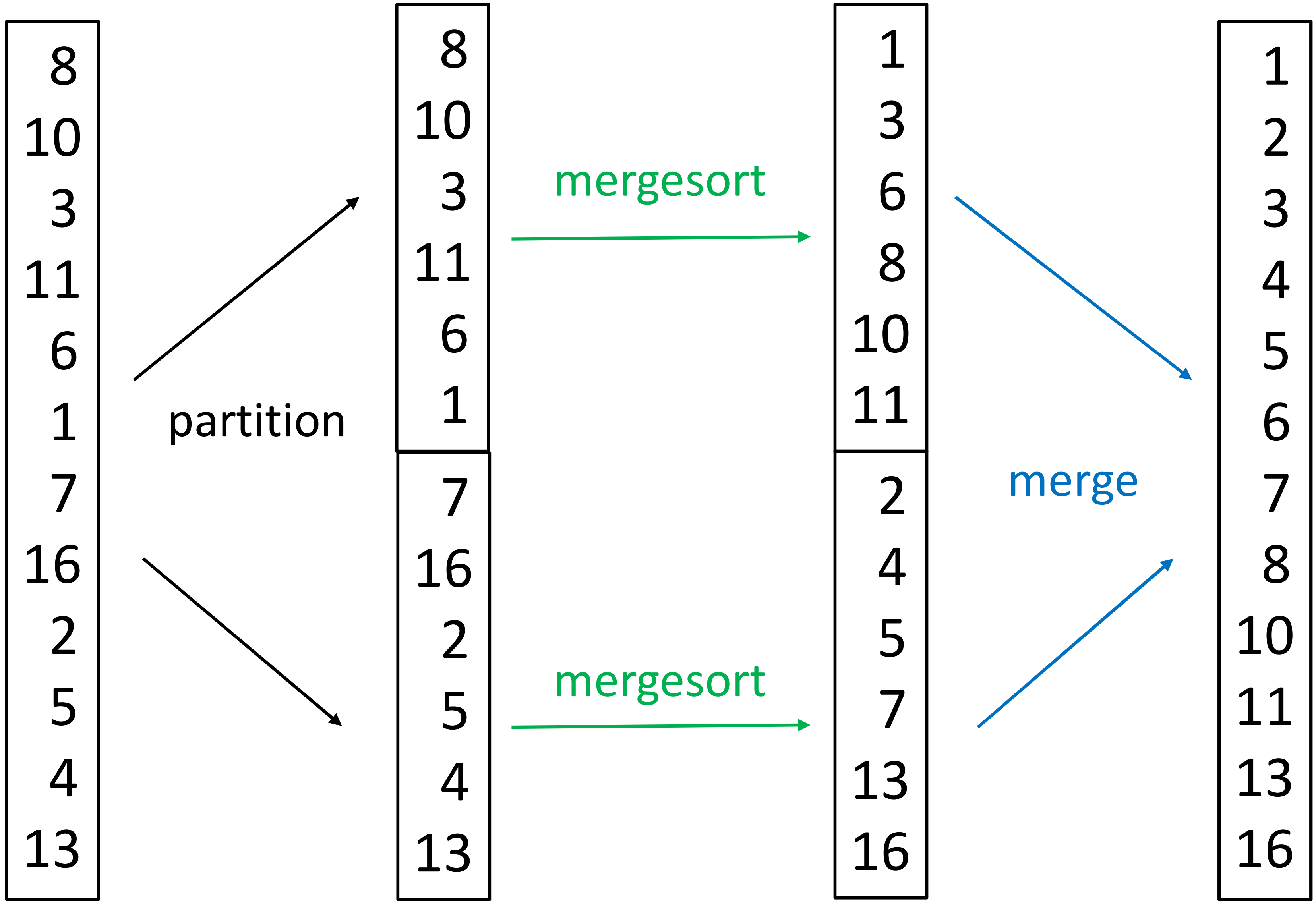
mergesort 2, quicksort

Recall: Mergesort

Given a list, partition it into two halves (1^{st} & 2^{nd}).

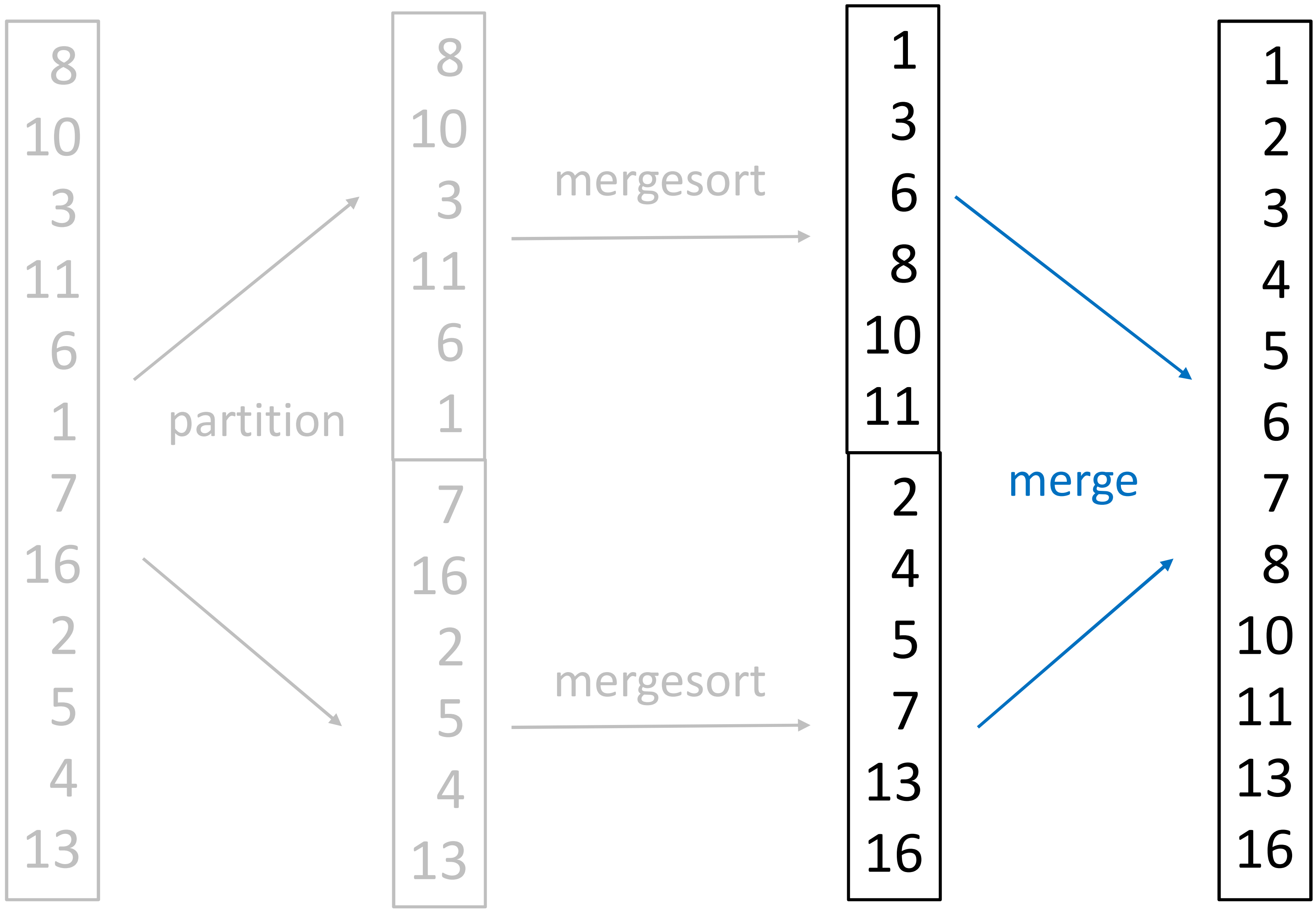
Sort each half (recursively).

Merge the two halves.


8
10
3
11
6
1
7
16
2
5
4
13
partition
8
10
3
11
6
1
7
16
2
5
4
13
mergesort
mergesort
1
3
6
8
10
11
2
4
5
7
13
16
merge
1
2
3
4
5
6
7
8
10
11
13
16

```
mergesort(list){
    if  list.length == 1              //  base case
        return list
    else{
        mid = (list.size - 1) / 2
        list1 =  list.getElements(0,mid)
        list2 =  list.getElements(mid+1, list.size-1)
        list1  =  mergesort(list1)
        list2  =  mergesort(list2)
        return   merge( list1, list2 )
    }
}
```

At the end of last lecture, we saw how merge worked.

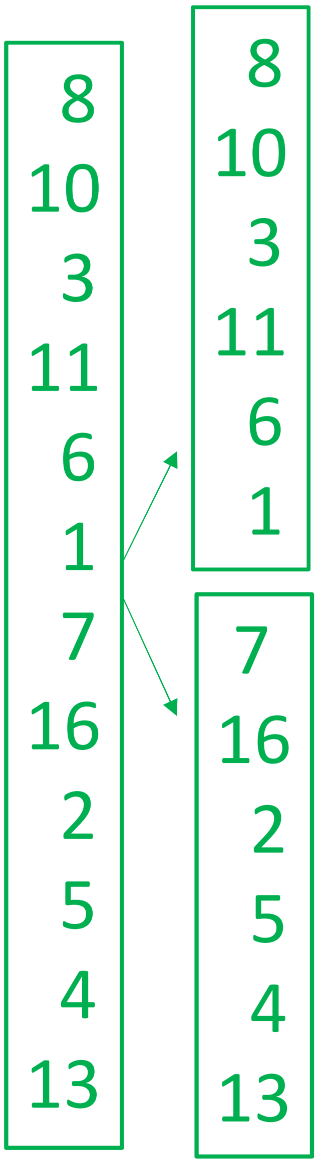


Let’s now consider the partitioning and recursion.

8
10
3
11
6
1
7
16
2
5
4
13

```
mergesort(list){
    if  list.length == 1
        return list
    else{
        mid = (list.size - 1) / 2
        list1 =  list.getElements(0,mid)
        list2 =  list.getElements(mid+1, list.size-1)
        list1  =  mergesort(list1)
        list2  =  mergesort(list2)
        return   merge( list1, list2 )
    }
}
```

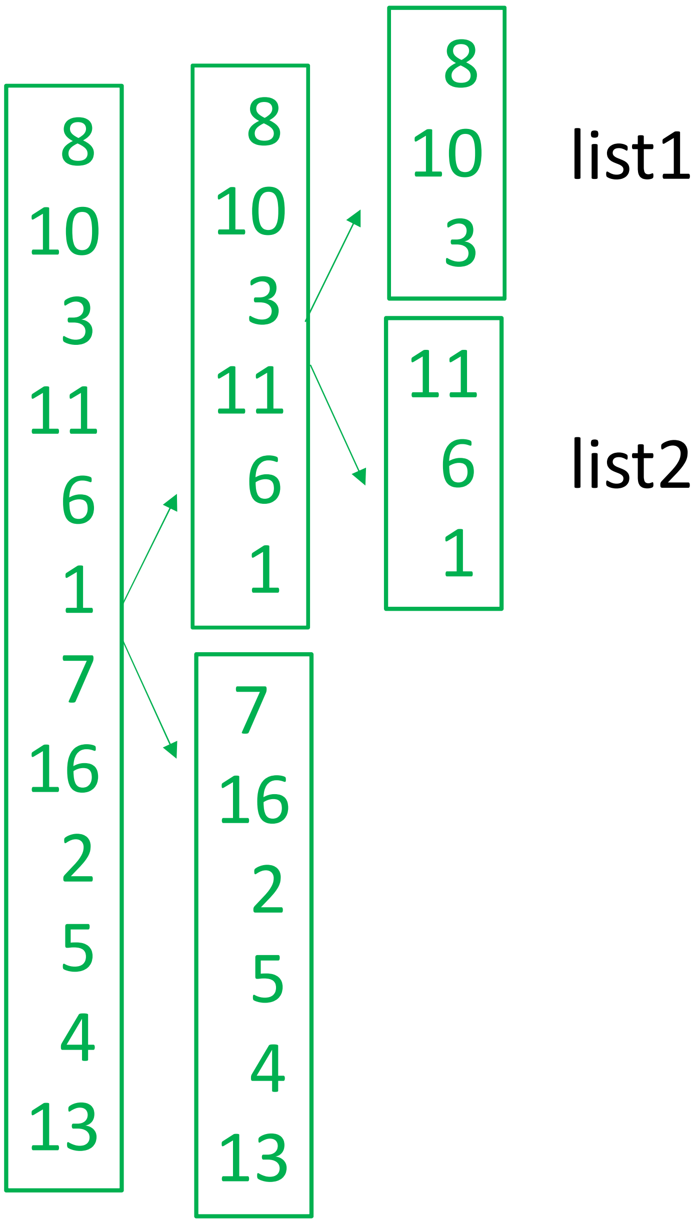
For each green rectangle there is one call to mergesort.

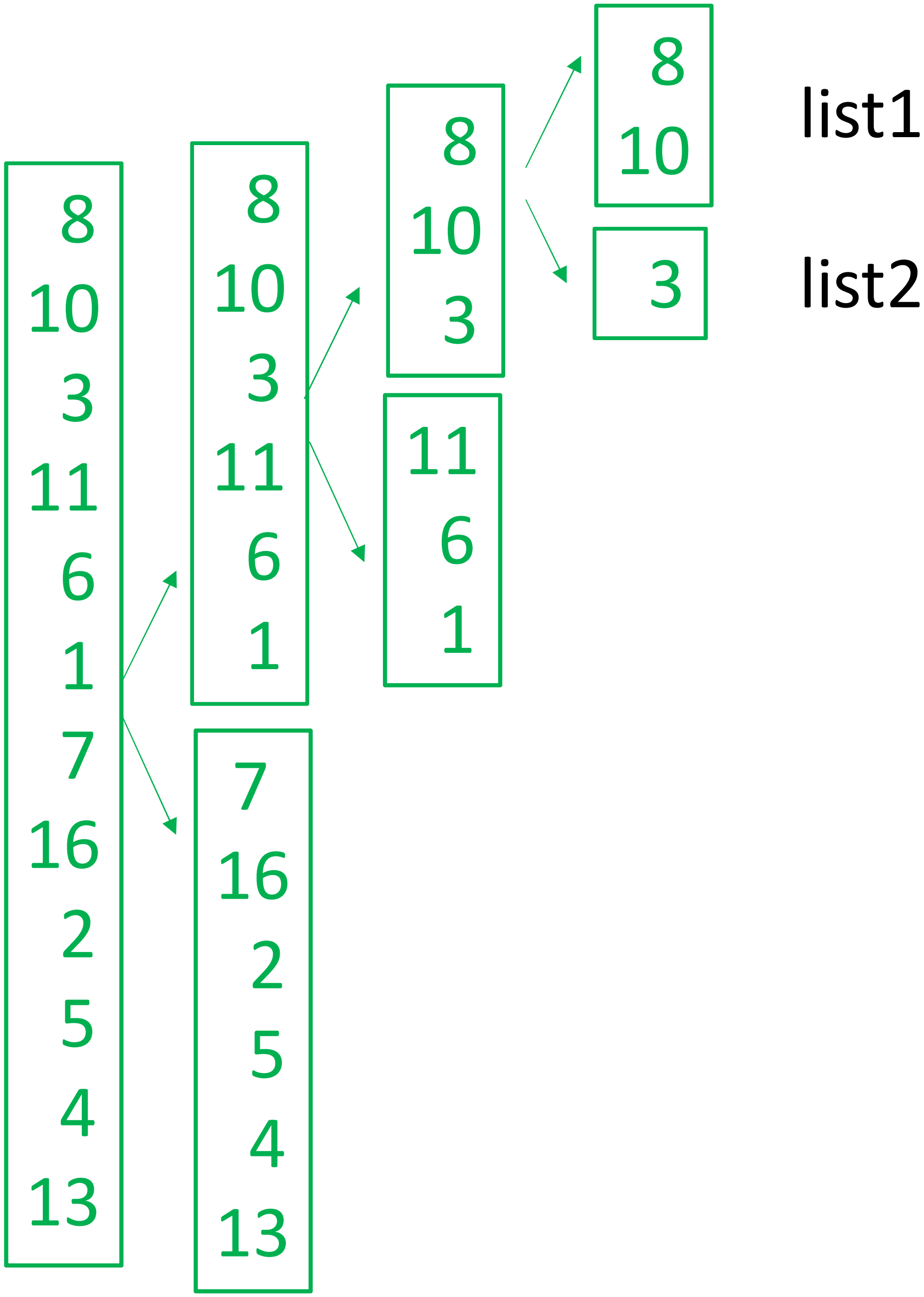


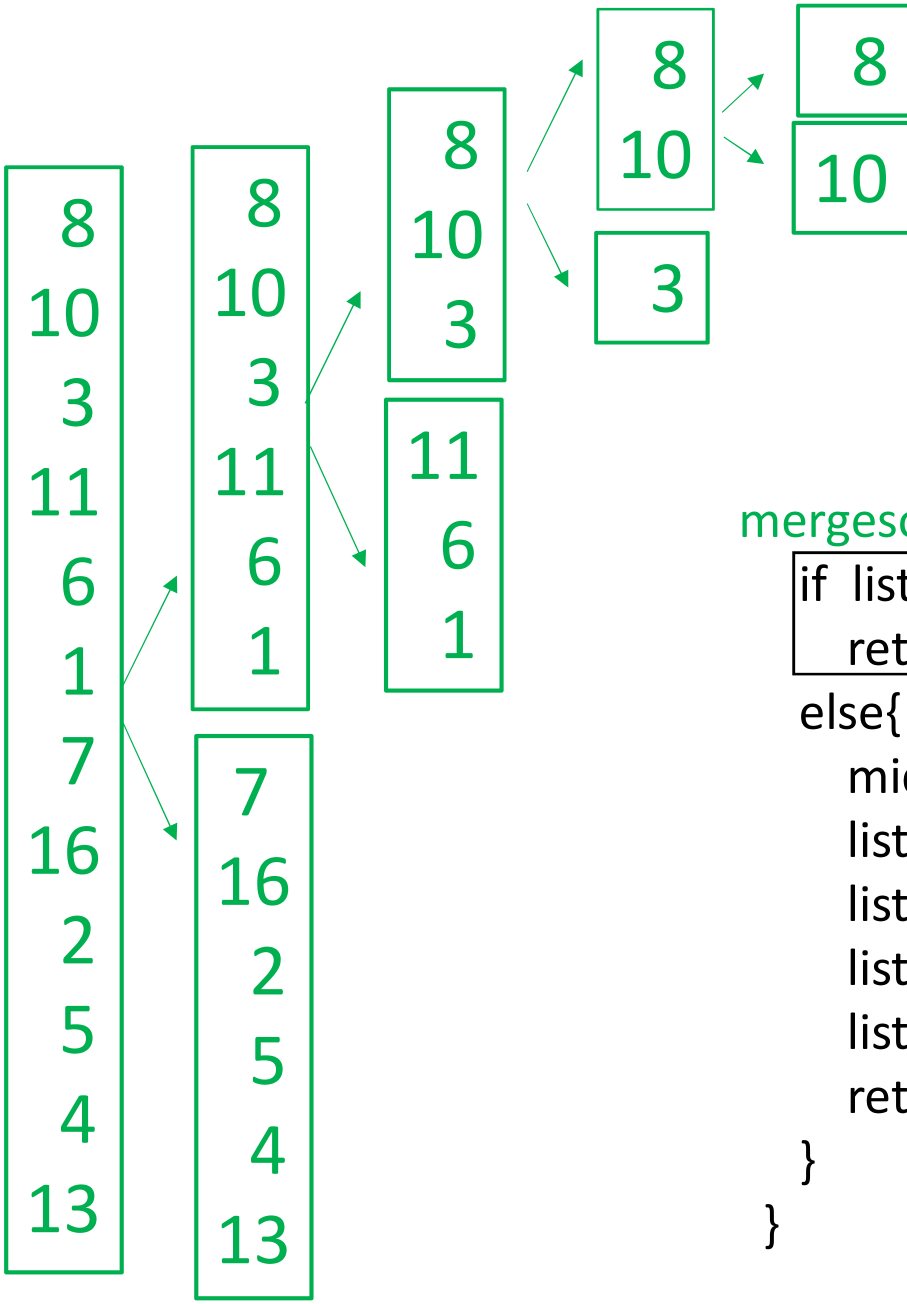
list1

list2

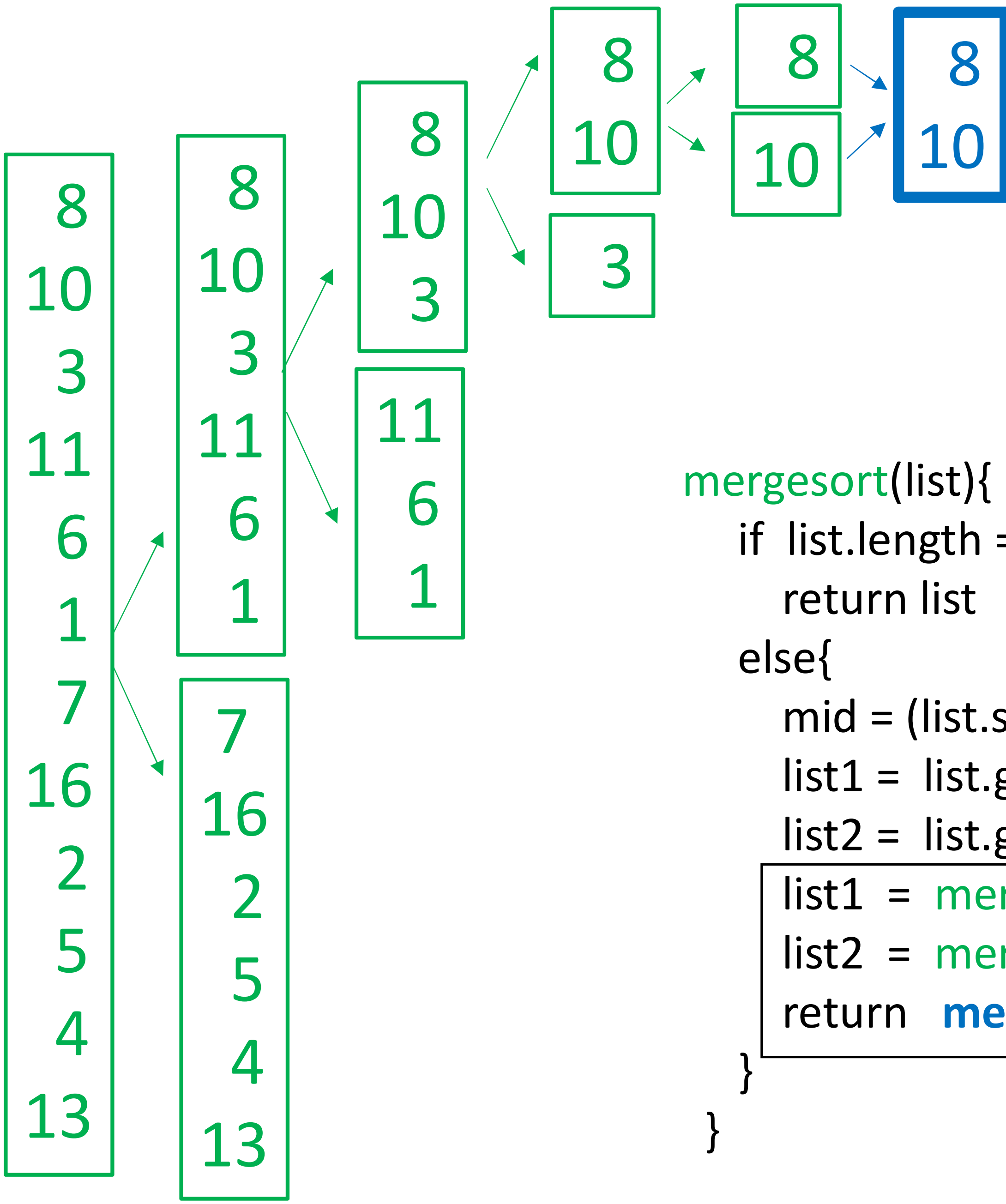
```
mergesort(list){
    if  list.length == 1
       return list
    else{
       mid = (list.size - 1) / 2
       list1 =  list.getElements(0,mid)
       list2 =  list.getElements(mid+1, list.size-1)
       list1  =  mergesort(list1)
       list2  =  mergesort(list2)
       return   merge( list1, list2 )
    }
  }
```


8
10
3
11
6
1
7
16
2
5
4
13
8
10
3
11
6
1
7
16
2
5
4
13
8
10
3
11
6
1
list1
list2

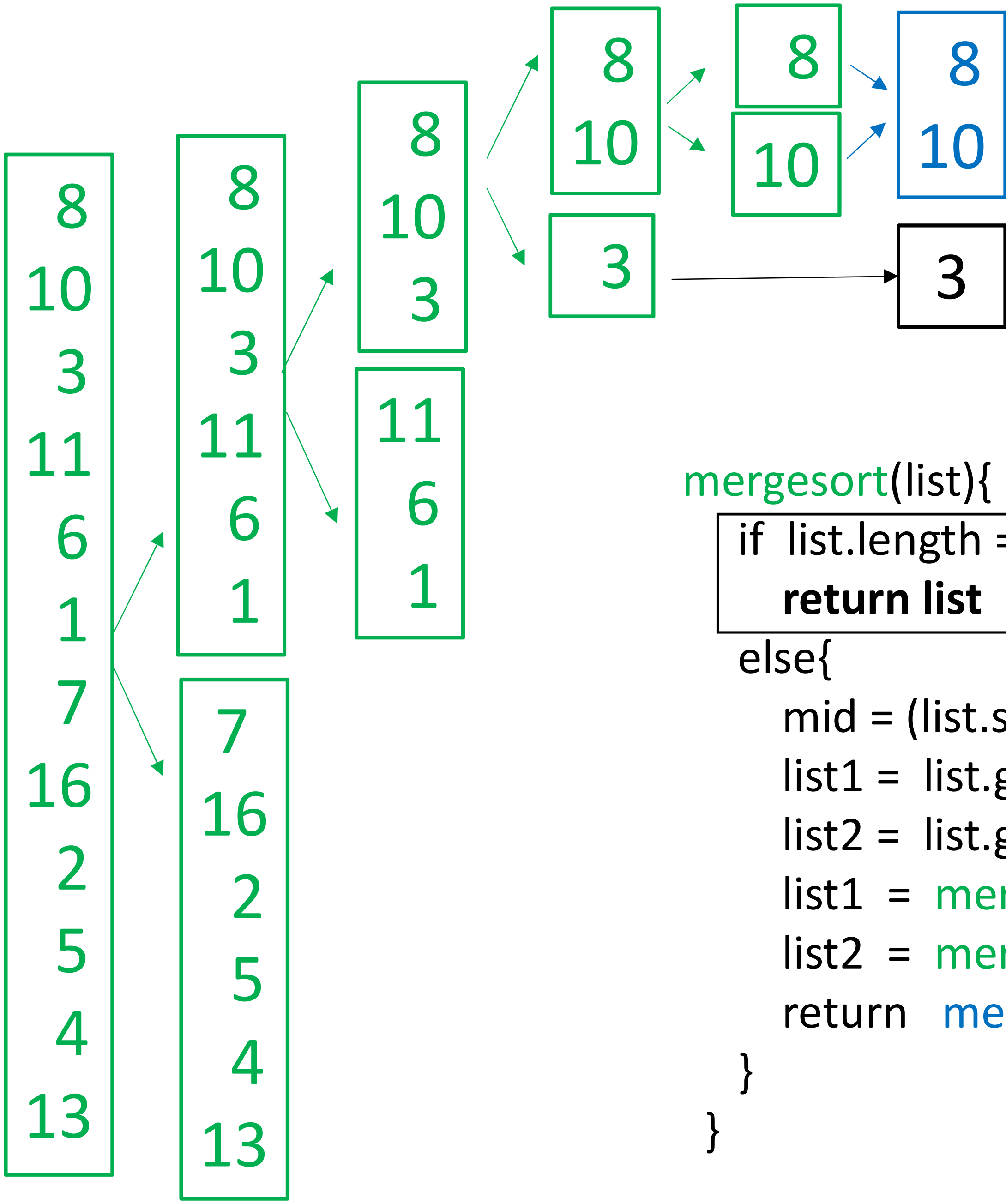

8
10
3
11
6
1
7
16
2
5
4
13
list1
list2



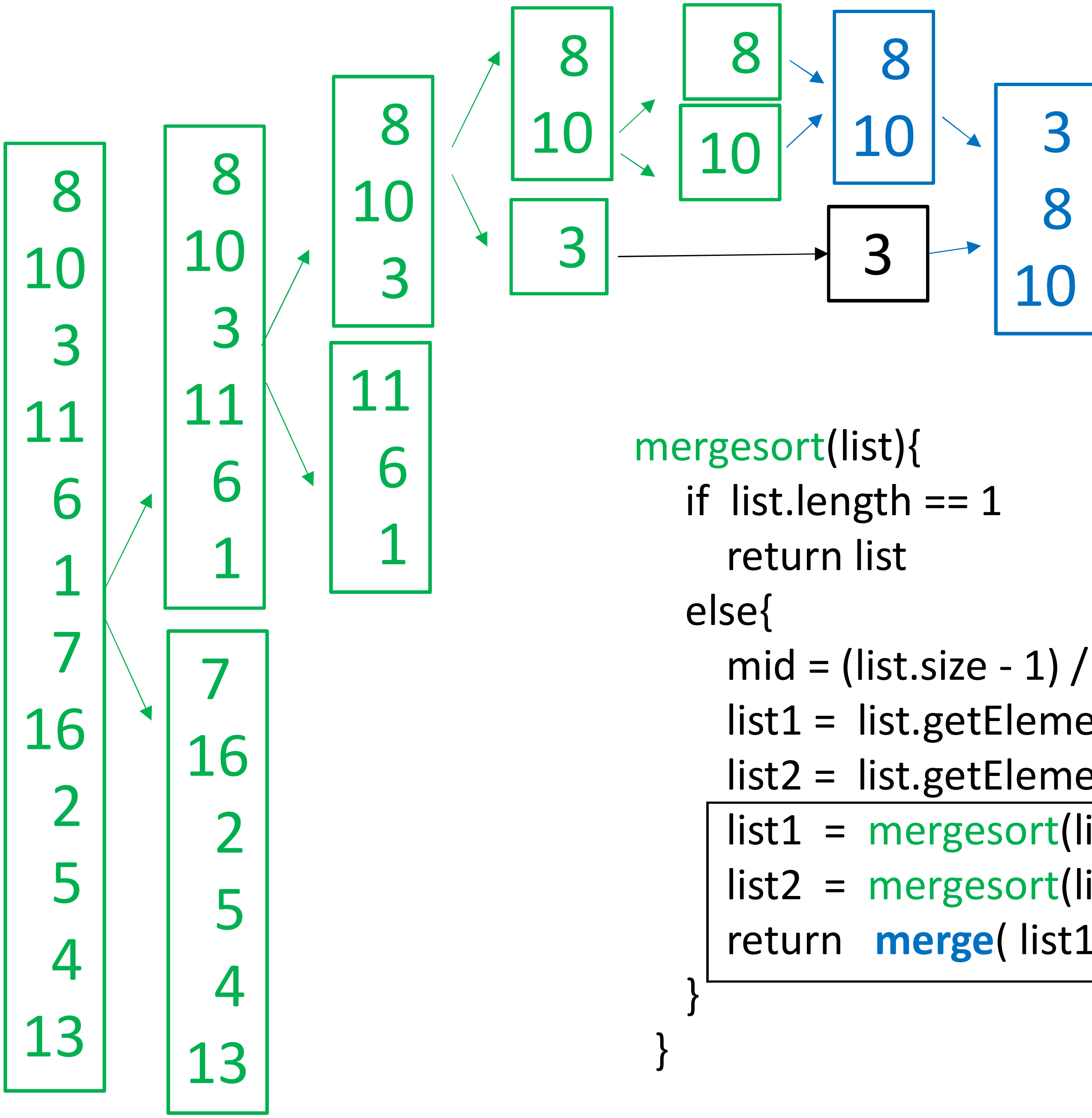

```
mergesort(list){
   if  list.length == 1
      return list
   else{
      mid = (list.size - 1) / 2
      list1 =  list.getElements(0,mid)
      list2 =  list.getElements(mid+1, list.size-1)
      list1  =  mergesort(list1)
      list2  =  mergesort(list2)
      return   merge( list1, list2 )
   }
}
```



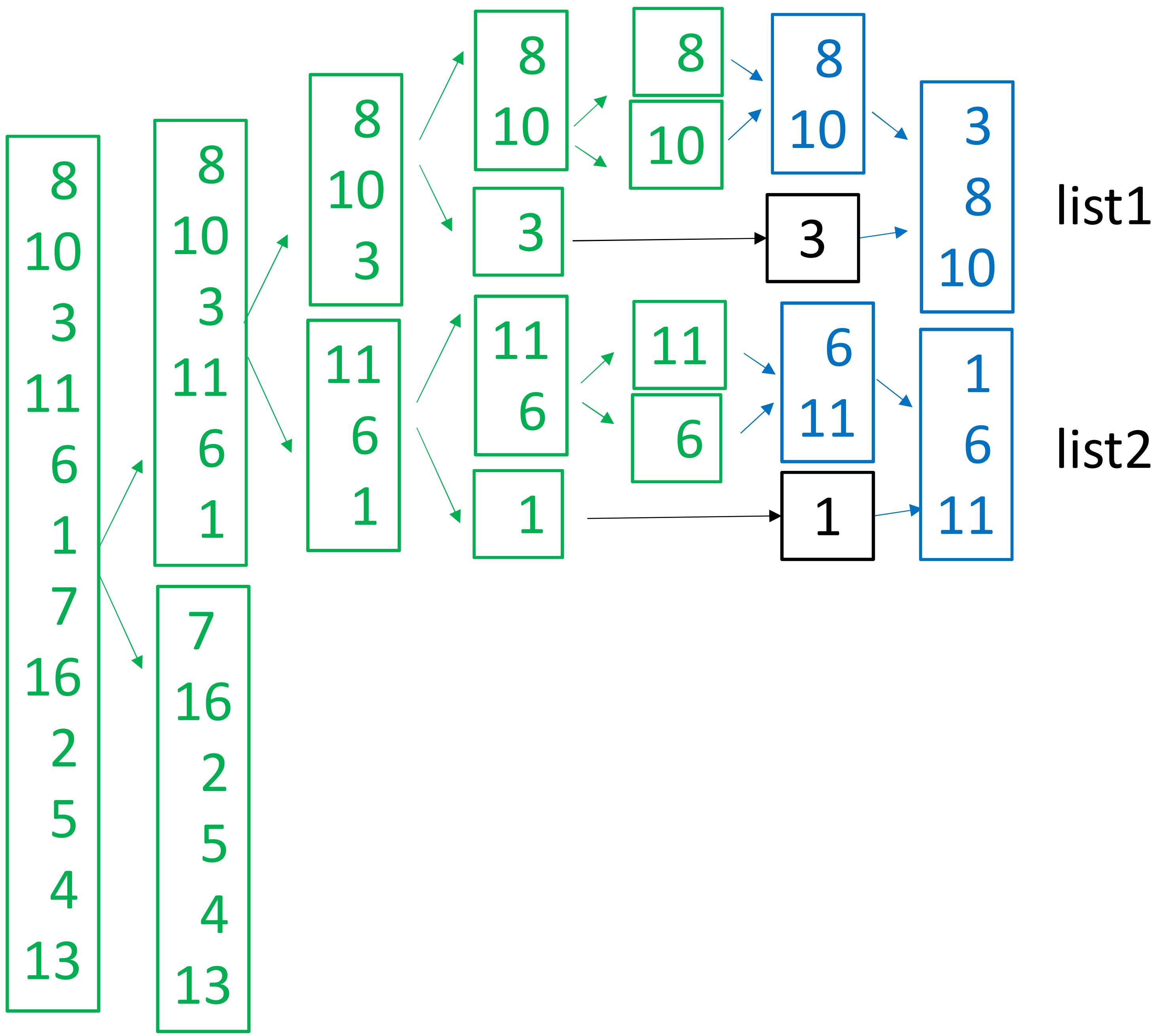

```
mergesort(list){
    if  list.length == 1
        return list
    else{
        mid = (list.size - 1) / 2
        list1 =  list.getElements(0,mid)
        list2 =  list.getElements(mid+1, list.size-1)
        list1  =  mergesort(list1)
        list2  =  mergesort(list2)
        return   merge( list1, list2 )
    }
}
```

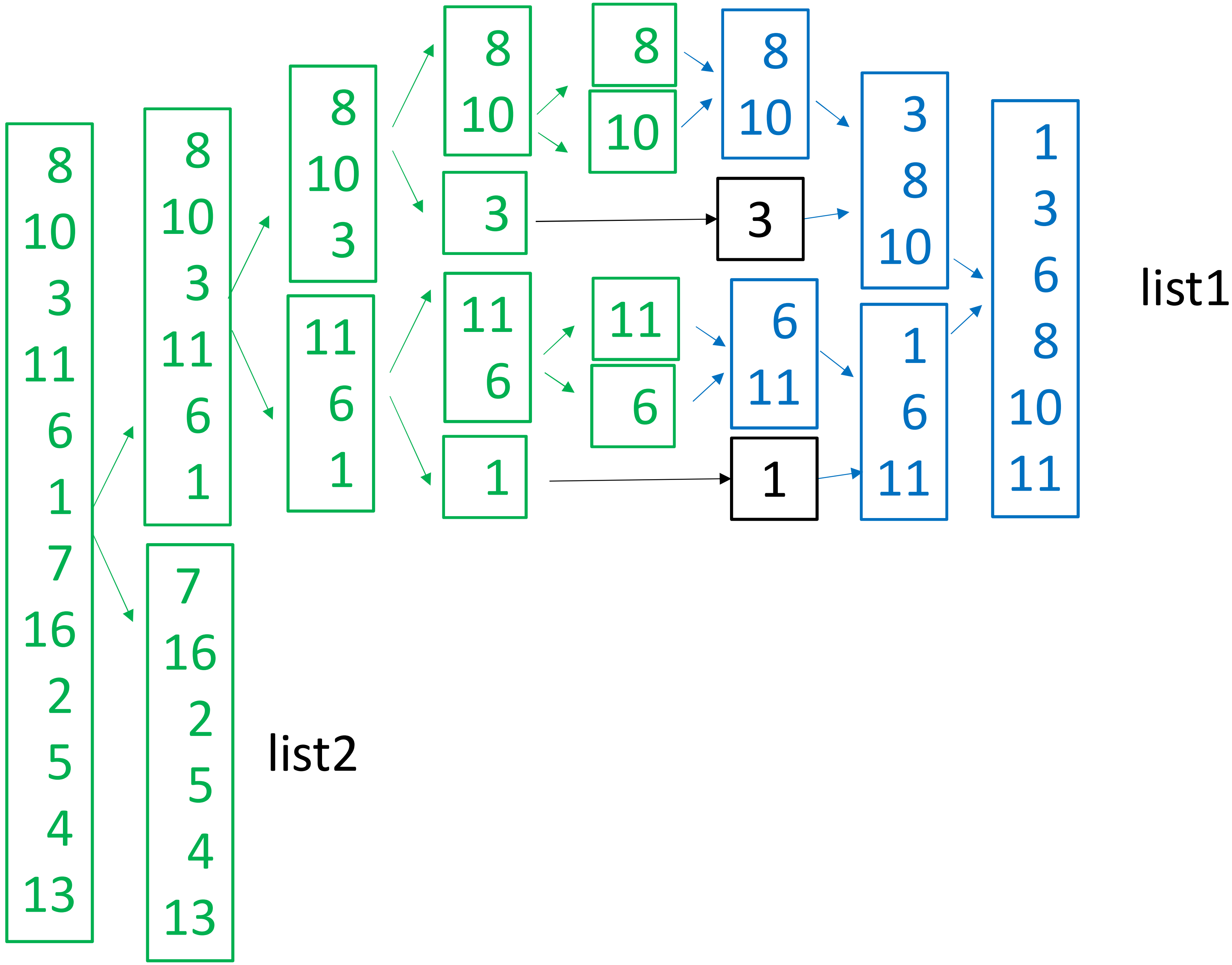


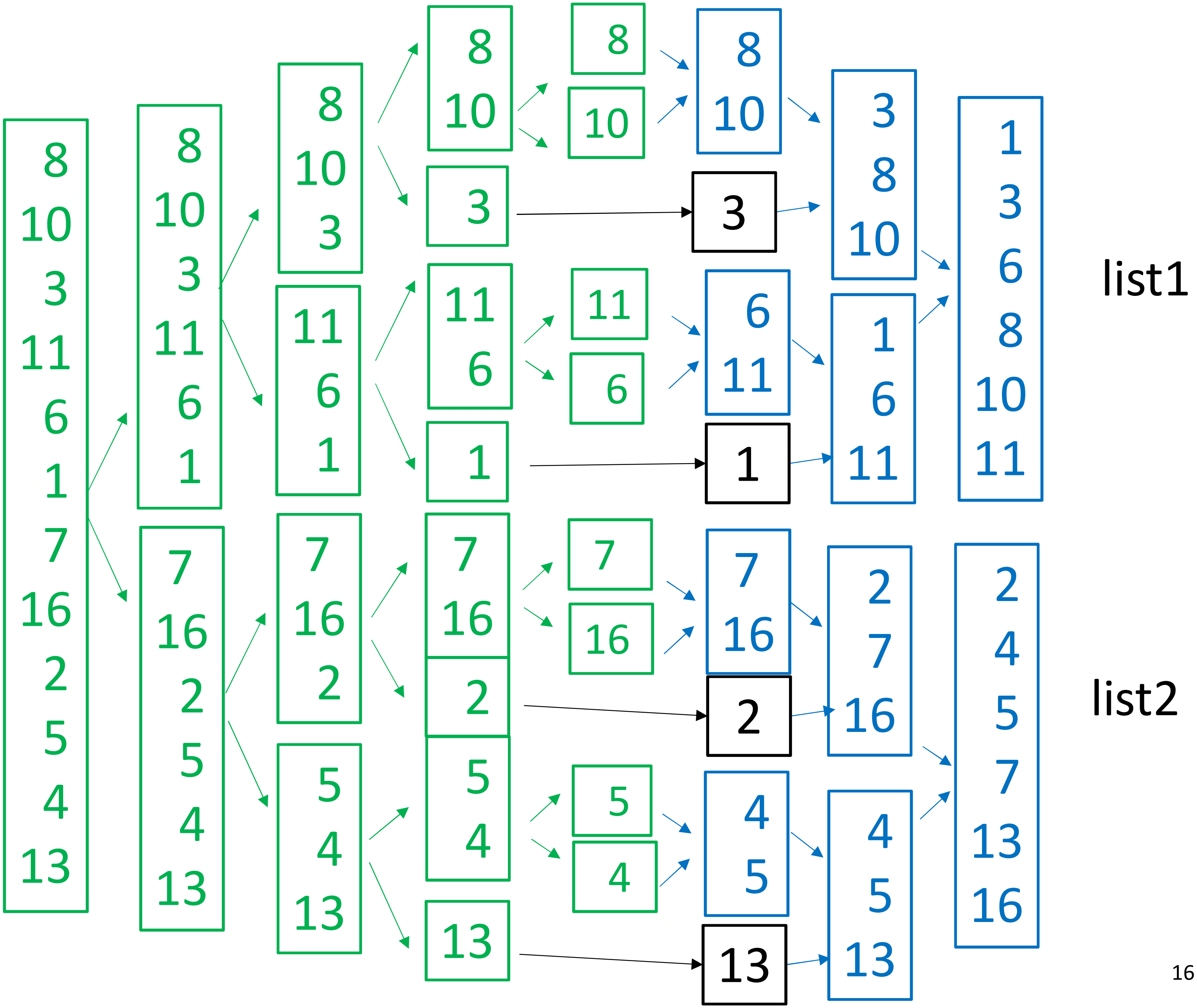

```
mergesort(list){
  if  list.length == 1
     return list
  else{
     mid = (list.size - 1) / 2
     list1 =  list.getElements(0,mid)
     list2 =  list.getElements(mid+1, list.size-1)
     list1  =  mergesort(list1)
     list2  =  mergesort(list2)
     return   merge( list1, list2 )
  }
}
```

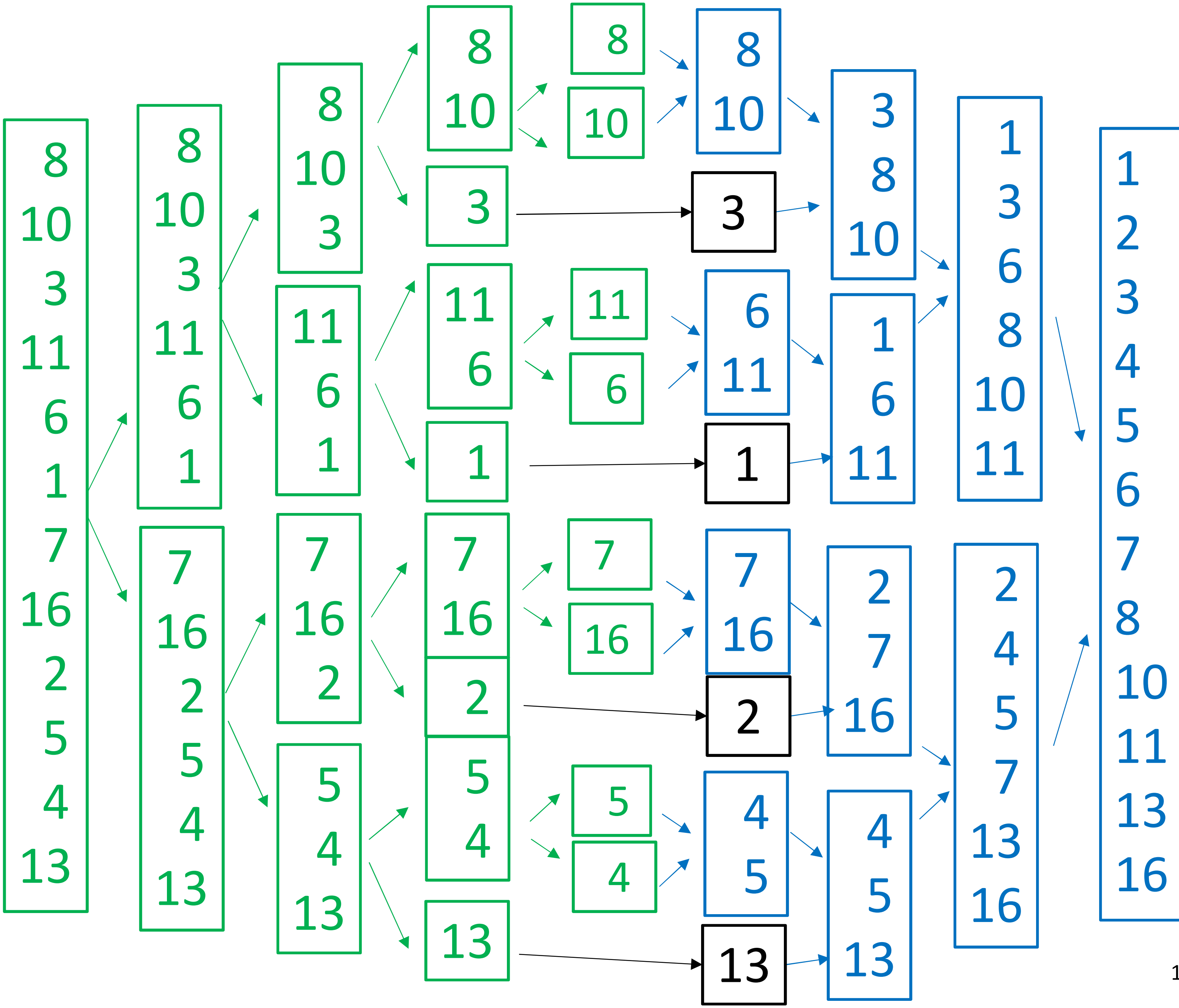



```
mergesort(list){
    if  list.length == 1
        return list
    else{
        mid = (list.size - 1) / 2
        list1 =  list.getElements(0,mid)
        list2 =  list.getElements(mid+1, list.size-1)
        list1  =  mergesort(list1)
        list2  =  mergesort(list2)
        return   merge( list1, list2 )
    }
}
```


8
10
3
11
6
1
7
16
2
5
4
13
3
8
10
list1
1
6
11
list2


8 10 3 11 6 1 7 16 2 5 4 13
8 10 3 11 6 1
7 16 2 5 4 13
list2
8 10 3
11 6 1
8 10
3
11 6
1
8
10
11
6
8 10
3
6 11
1
3 8 10
1 6 11
1 3 6 8 10 11
list1

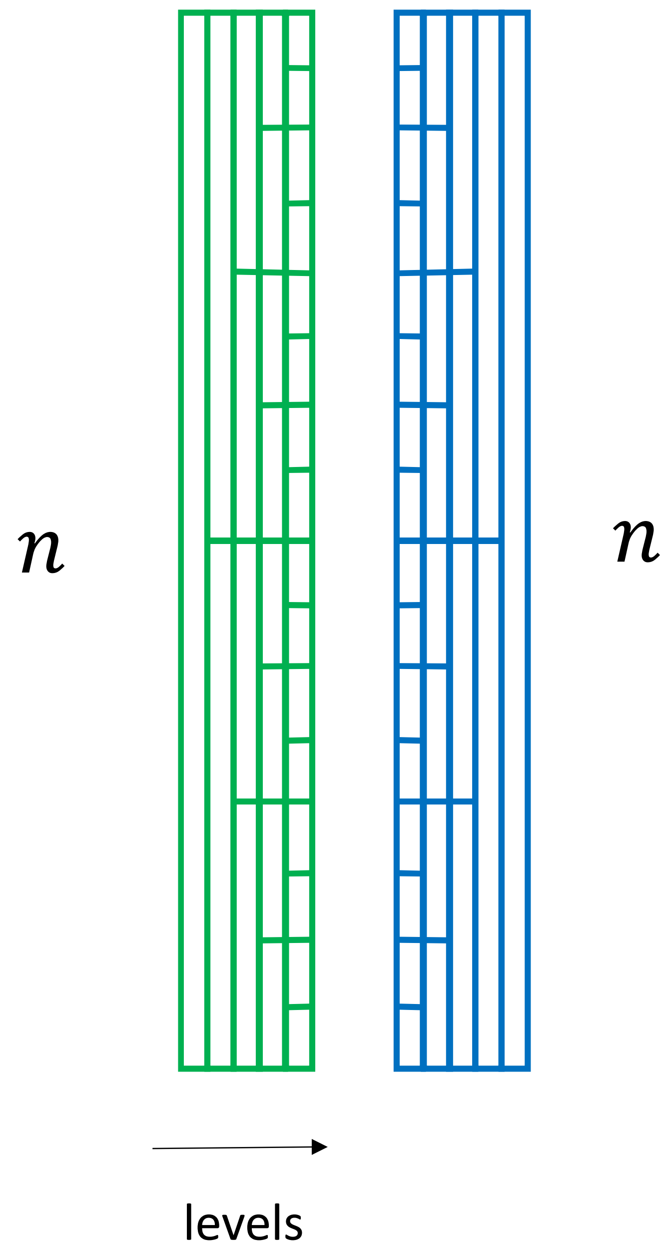

list1
list2



8 10 3 11 6 1 7 16 2 5 4 13
8 10 3 11 6 1
7 16 2 5 4 13
8 10 3
11 6 1
7 16 2
5 4 13
8 10
3
11 6
1
7 16
2
5 4
13
8
10
11
6
7
16
5
4
8 10
3
6 11
1
7 16
2
4 5
13
3 8 10
1 6 11
2 7 16
4 5 13
1 3 6 8 10 11
2 4 5 7 13 16
1 2 3 4 5 6 7 8 10 11 13 16

Q: How many levels ?

A:



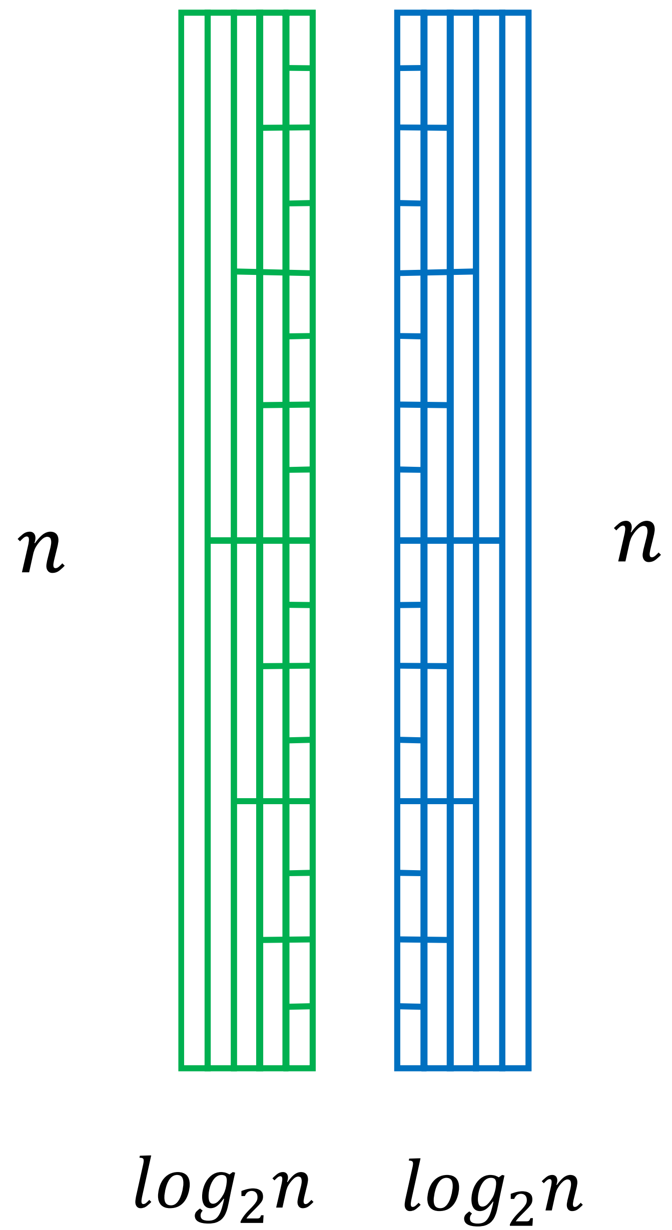
Q: How many levels ?

A: $\sim 2\ log_2 n$

Why?

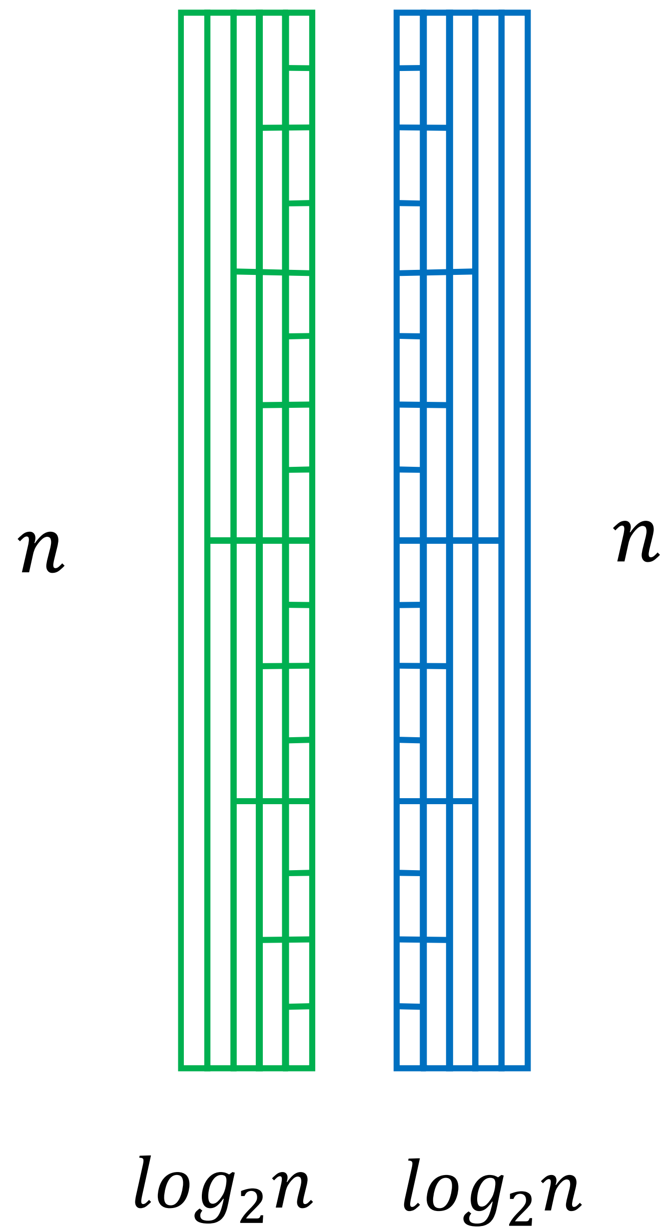
Because at each green level, we partition each list into two ~equal size sublists (halving).

At each blue level, we do the opposite, namely we merge two lists of approximately equal size (doubling).



Q: How many operations are required to mergesort a list of size n ?

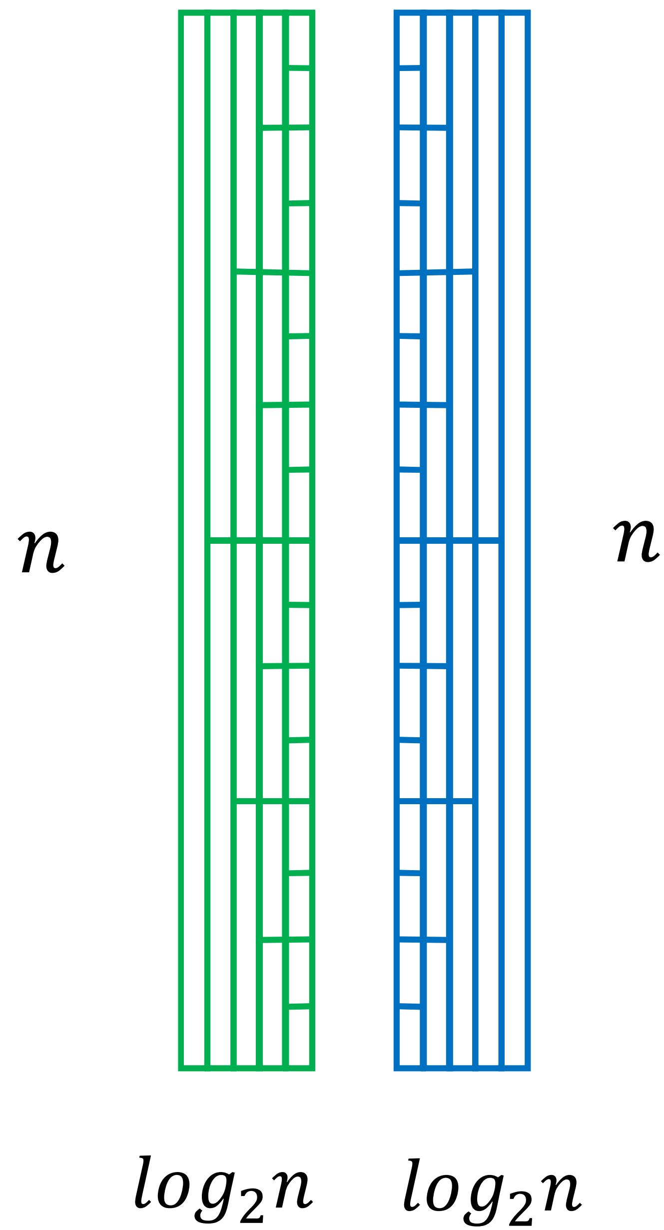
A:



Q: How many operations are required to mergesort a list of size n ?

A: $O(\ n \ \ log_2 n\ )$

For each level, each of the n elements is either put into a new list or merged with a bigger list. So there is $O(n)$ work at each level and $2\ log_2 n$ levels.



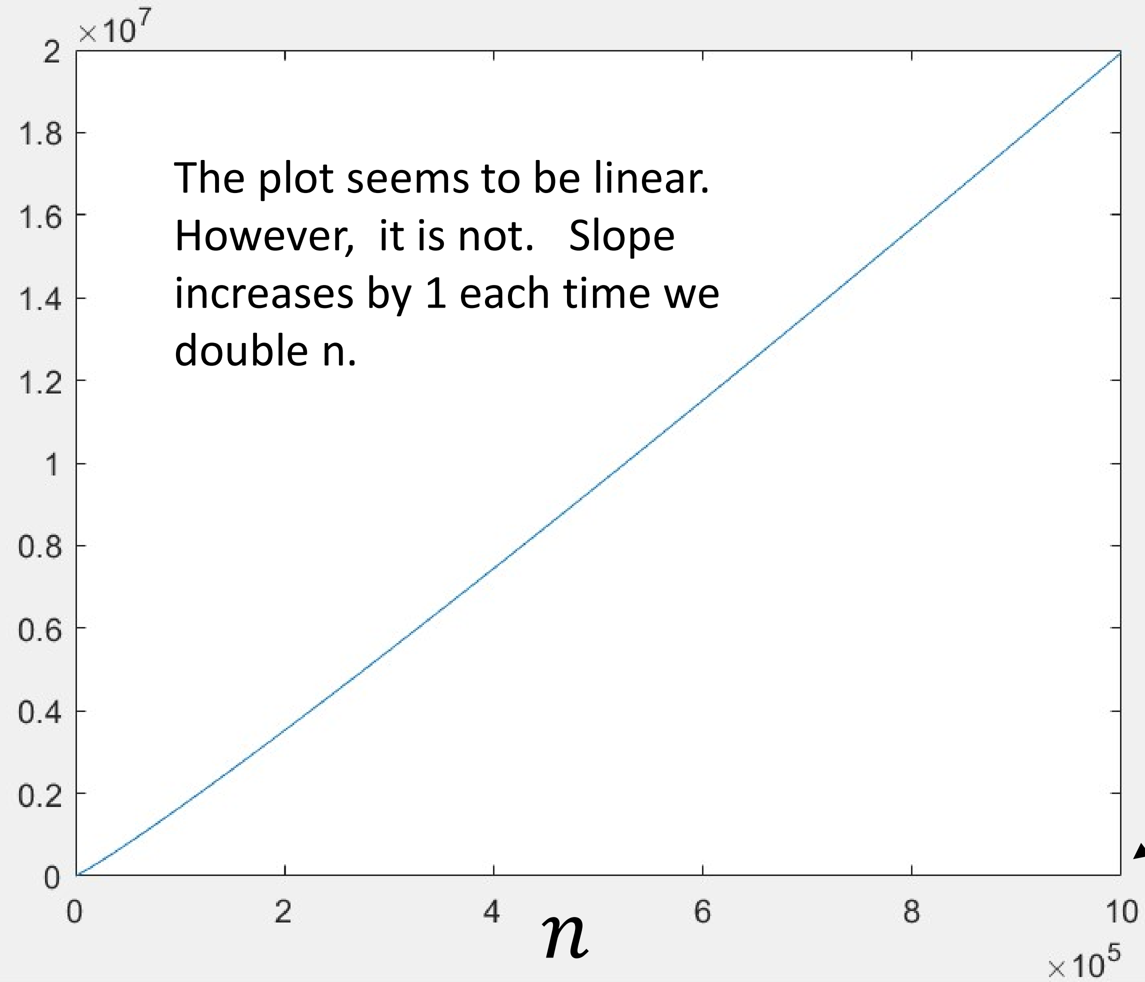
$n \log_2 n$ is much closer to n than to n^2

$\log_2 n$	n	$\boldsymbol{n \log_2 n}$	n^2
10	$2^{10} \approx 10^3$	$\mathbf{10^4}$	10^6
20	$2^{20} \approx 10^6$	$\mathbf{10^7}$	10^{12}
30	$2^{30} \approx 10^9$	$\mathbf{10^{10}}$	10^{18}

Plot of $n\ log_2 n$ vs. n

$\mathbf{17 * 10^6}$

$n\ log_2 n$



The plot seems to be linear. However, it is not. Slope increases by 1 each time we double n.

n

$10 * 10^5$
$= \mathbf{10^6}$
$\approx 2^{17}$

$$O(n) \quad < \quad O(n \ log_2 n) \quad \ll \quad O(n^2)$$

mergesort
quicksort
heapsort

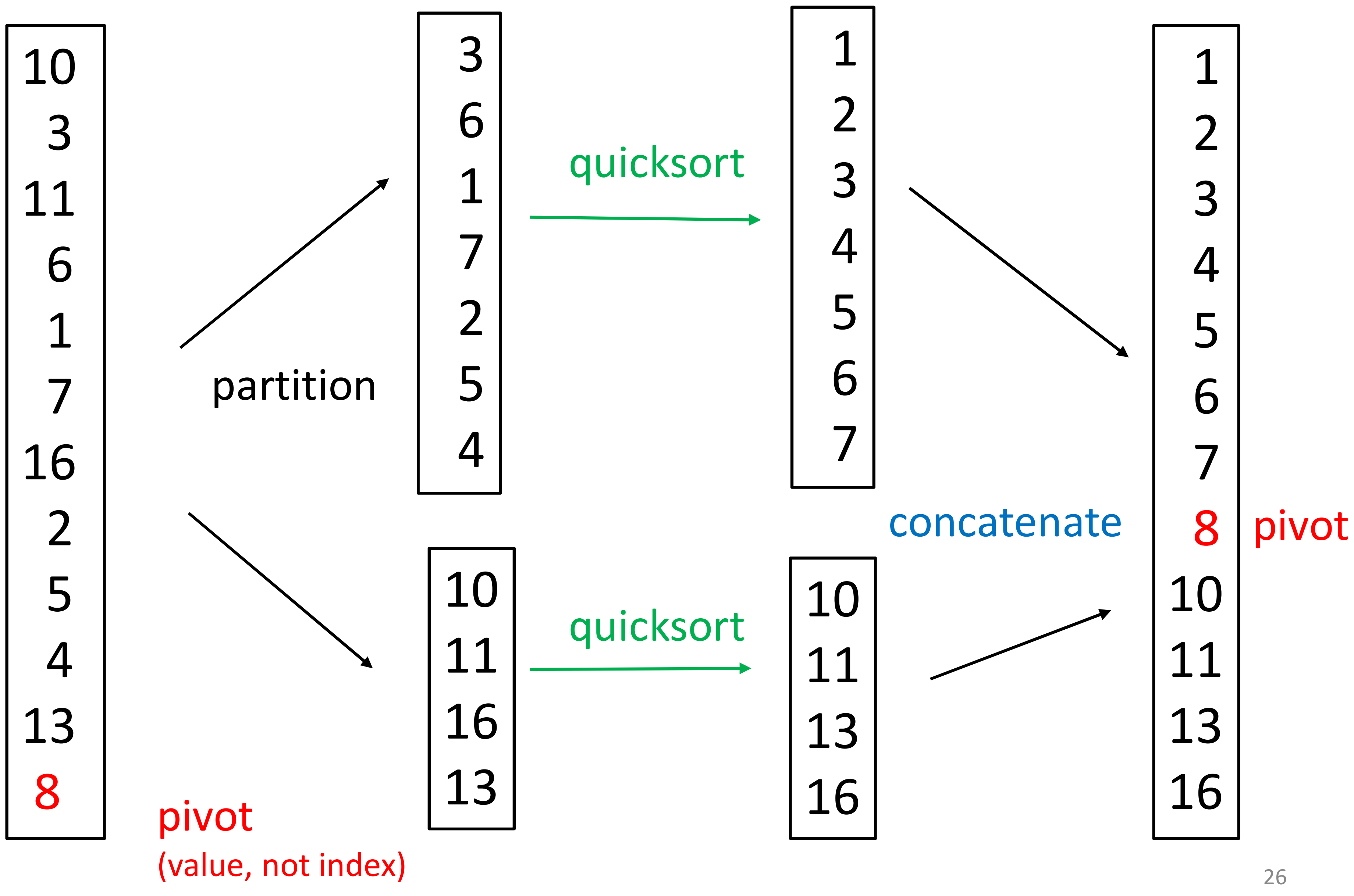
bubble sort
selection sort
insertion sort

COMP 250

Lecture 21

mergesort 2, quicksort

Oct. 27, 2021

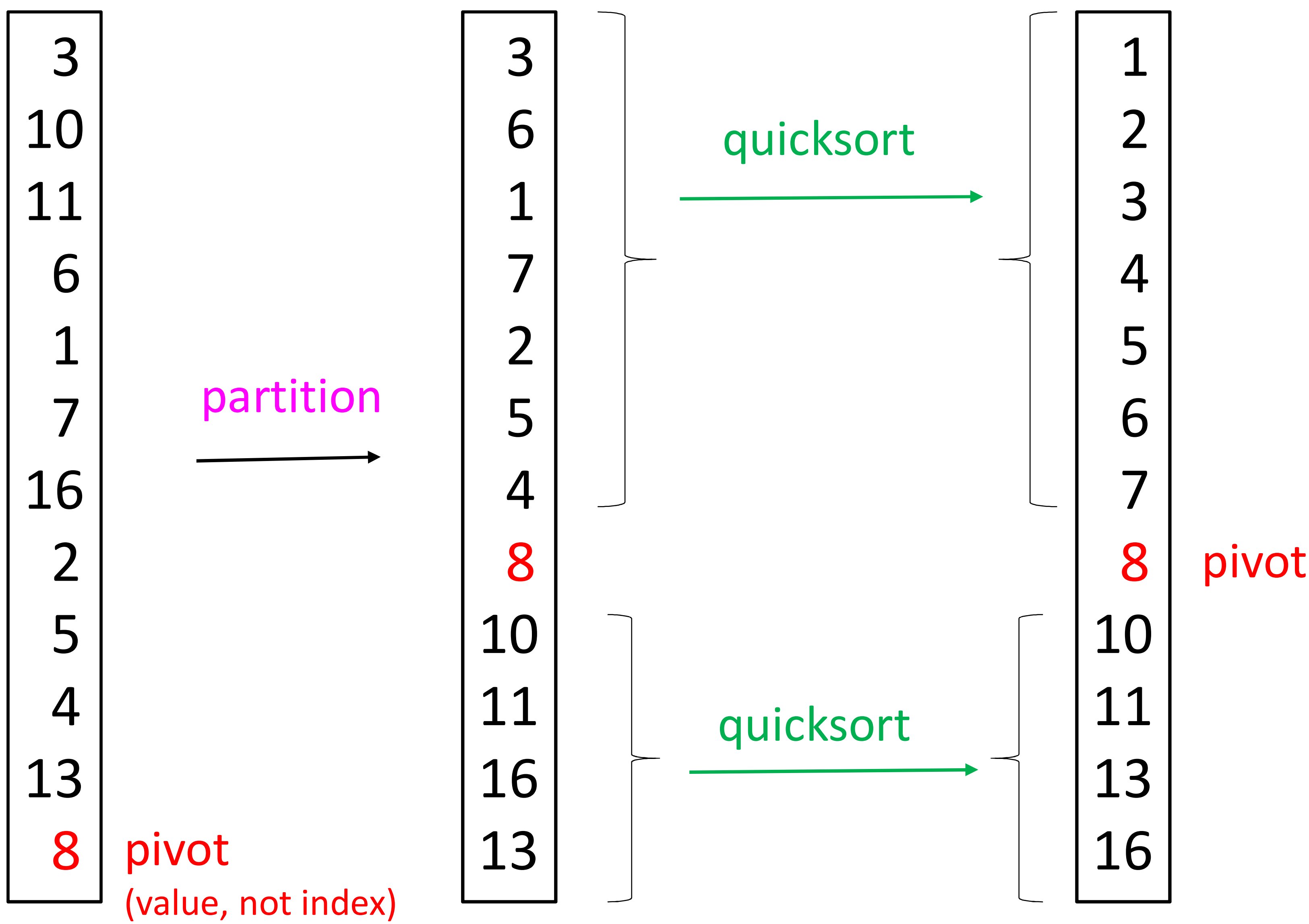

10
3
11
6
1
7
16
2
5
4
13
8
pivot
(value, not index)
partition
3
6
1
7
2
5
4
10
11
16
13
quicksort
quicksort
1
2
3
4
5
6
7
10
11
13
16
concatenate
1
2
3
4
5
6
7
8
pivot
10
11
13
16

Quicksort

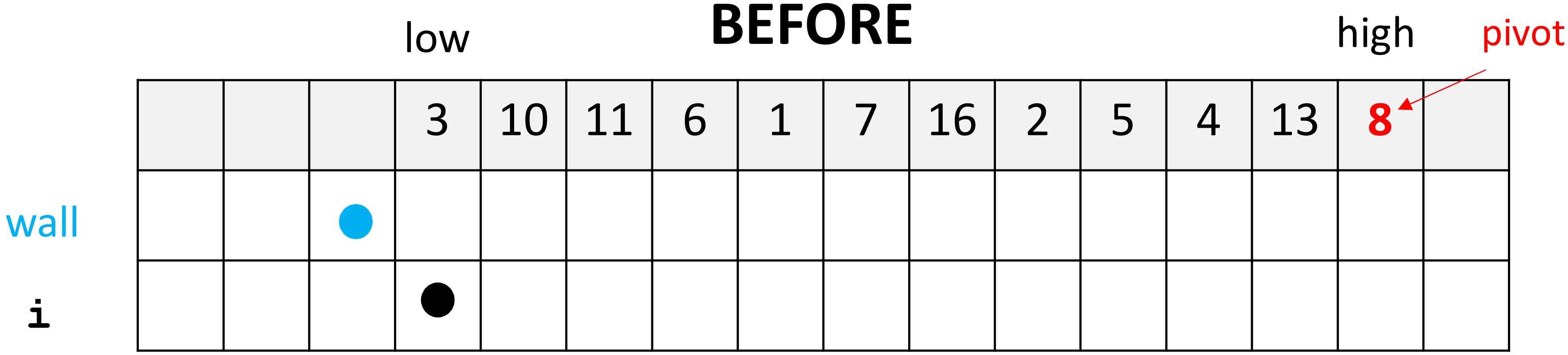
```
quicksort(list){
   if  list.length <= 1          //   base case
      return list
   else{
      pivot =  list.getLast()   // or some other element
      list1  =   list.getElementsLessThan(pivot)
      list2  =   list.getElementsGreaterOrEqual(pivot)
      list1 = quicksort(list1)
      list2 = quicksort(list2)
      return  concatenate( list1, pivot, list2 )
   }
}
```

Quicksort is quick because it can be done "in place" (using an array)

```
quicksort(list, low, high ){      //  doesn't return anything
    if  low < high  {
       wall  =   partition (list, low, high)
       quicksort(list, low, wall - 1)
       quicksort(list, wall + 1,  high)
     }
}
//    list elements are reordered but size doesn't change
```

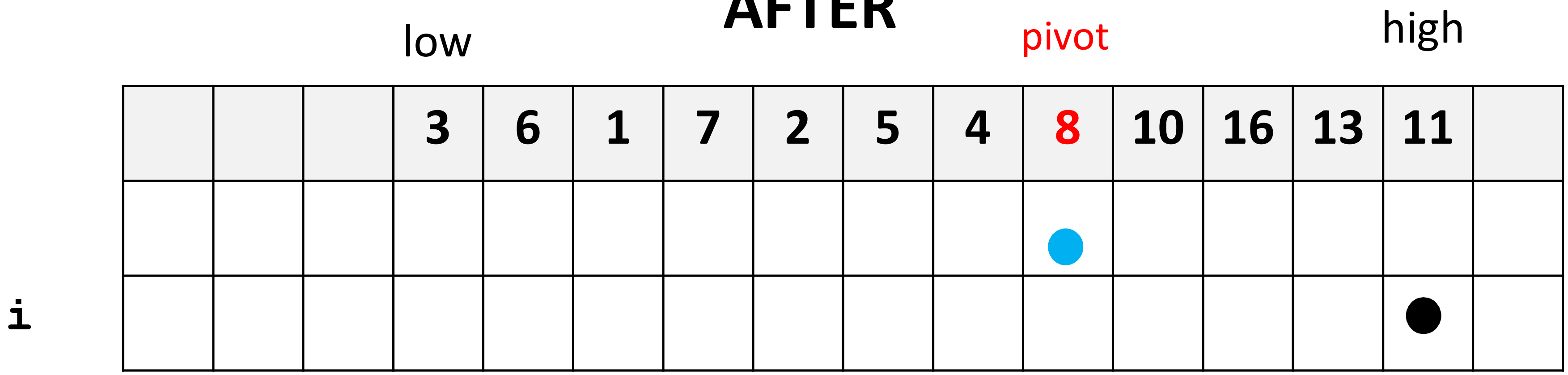

3
10
11
6
1
7
16
2
5
4
13
8
pivot
(value, not index)
partition
3
6
1
7
2
5
4
8
10
11
16
13
quicksort
quicksort
1
2
3
4
5
6
7
8
pivot
10
11
13
16

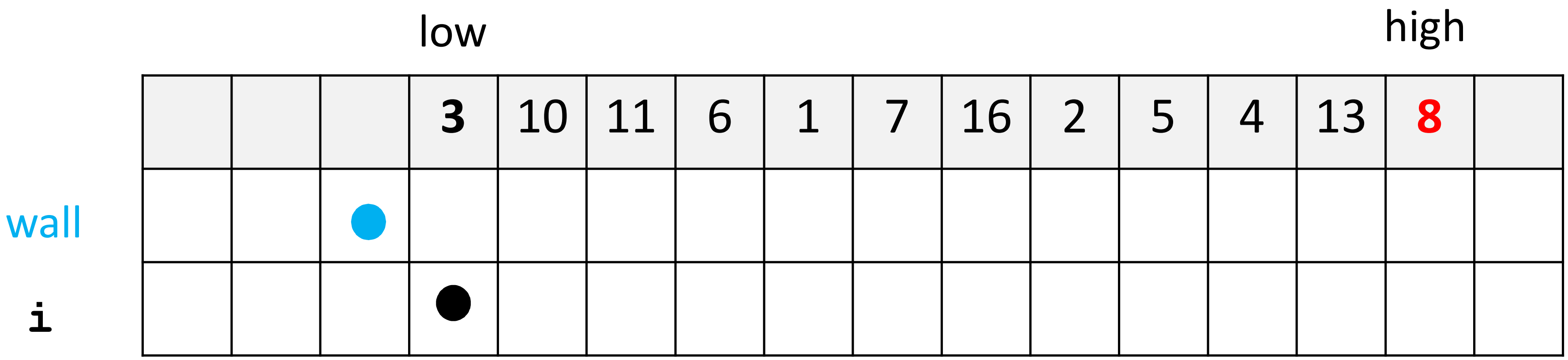
BEFORE



In the partition algorithm, we increment an index `i` and a wall index. Elements that are less than (or equal to) pivot are swapped such that are to the left of the wall.

AFTER






```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)

        }
   return wall
}
```

				low											high	
				3	10	11	6	1	7	16	2	5	4	13	**8**	
wall			●													
i				●												

```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
```

... then list[**i**] should end up to the left of the pivot

(or at the pivot, in the case that list[**i**] == pivot)

```
      }
  return wall
}
```

				low										high	
				3	10	11	6	1	7	16	2	5	4	13	**8**	
wall			●													
i				●												

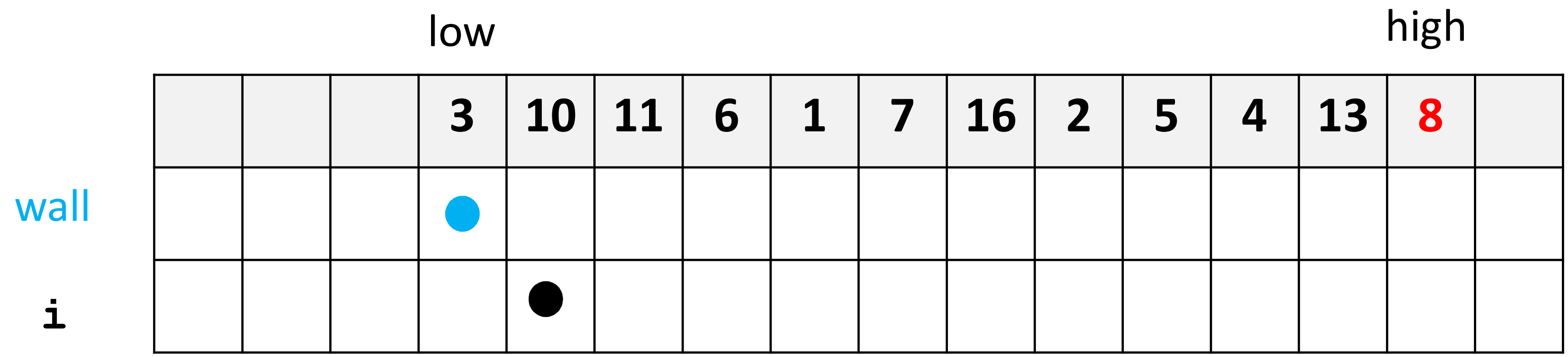
```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                list.swap(wall, i)
      }
   return wall
}
```

3 <= 8 so we enter the block.
(wall is incremented, swap does nothing)

				low											high	
				3	10	11	6	1	7	16	2	5	4	13	8	
wall				●												
i				●												

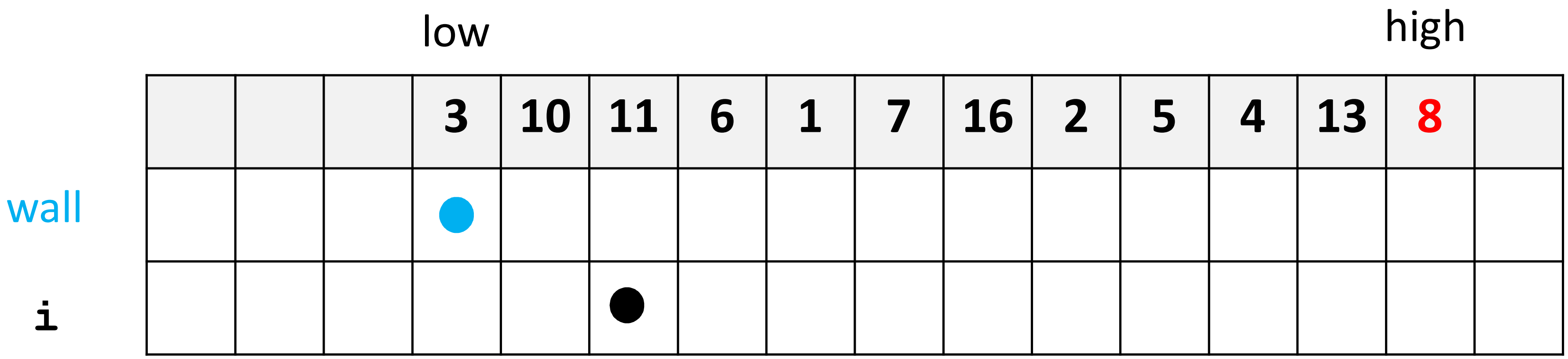
```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
         if ( list[i]  <=  pivot ){
            wall ++
            if  (wall != i)
                  list.swap(wall, i)
         }
    return wall
}
```

At the end of the block, the situation is shown above.



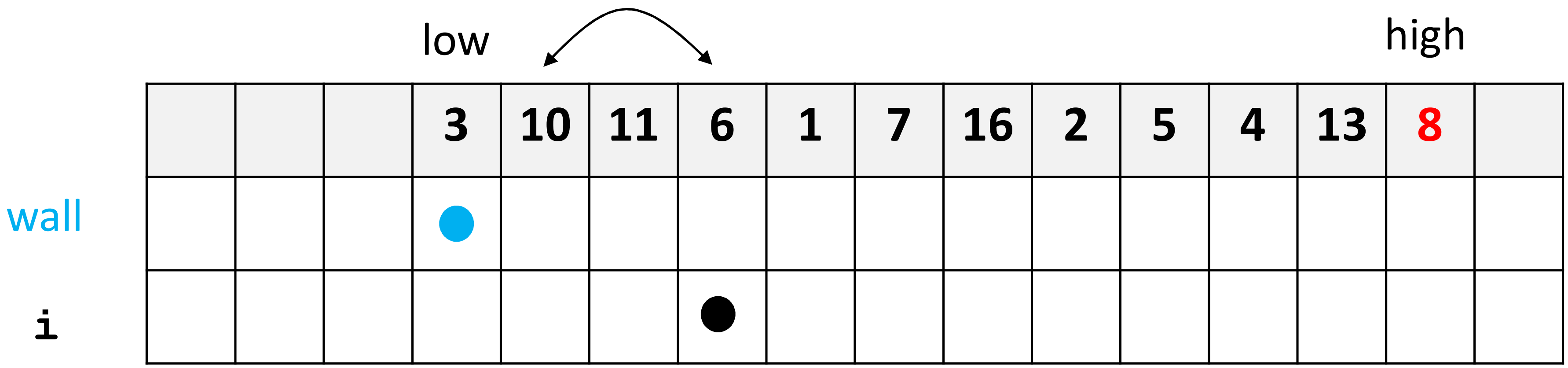

```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                  list.swap(wall, i)
      }
   return wall
}
```

10 > 8 so we don’t enter the block




```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                  list.swap(wall, i)
      }
   return wall
}
```

11 > 8 so we don't enter the block



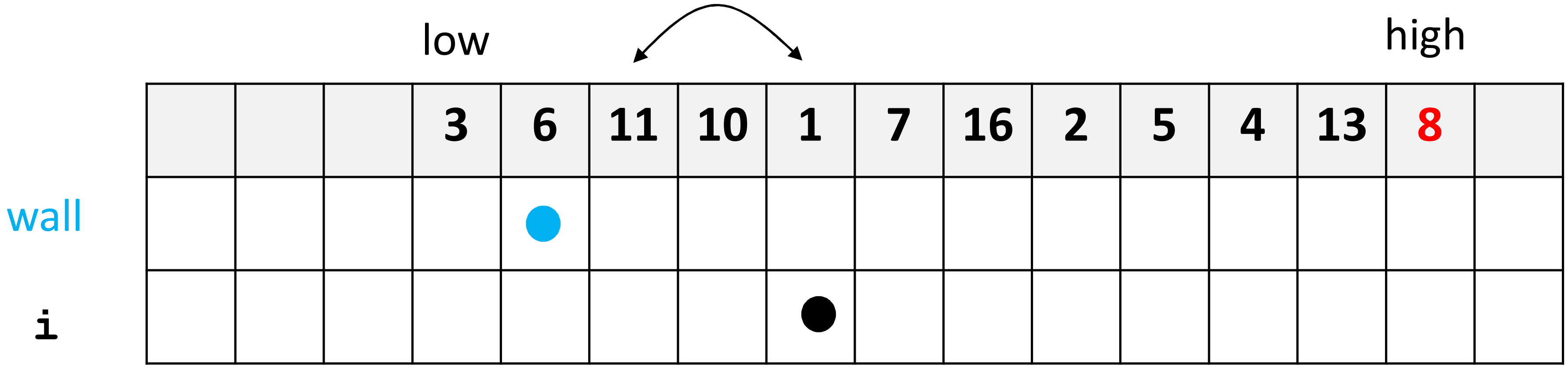

```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
         if ( list[i]  <=  pivot ){
              wall ++
              if  (wall != i)
                    list.swap(wall, i)
         }
    return wall
}
```

6 <= 8 so we enter the block.
Increment wall and swap 10 and 6.

				low											high	
				3	**6**	**11**	**10**	**1**	**7**	**16**	**2**	**5**	**4**	**13**	**8**	
wall					●											
`i`							●									

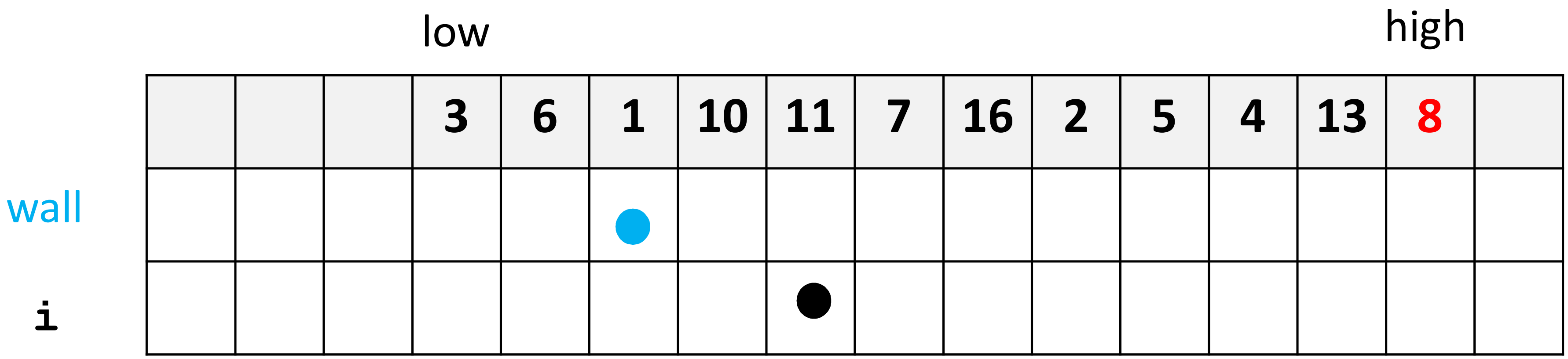
```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
         if ( list[i]  <=  pivot ){
            wall ++
            if  (wall != i)
                     list.swap(wall, i)
         }
    return wall
}
```

At the end of the block, the situation is shown above.



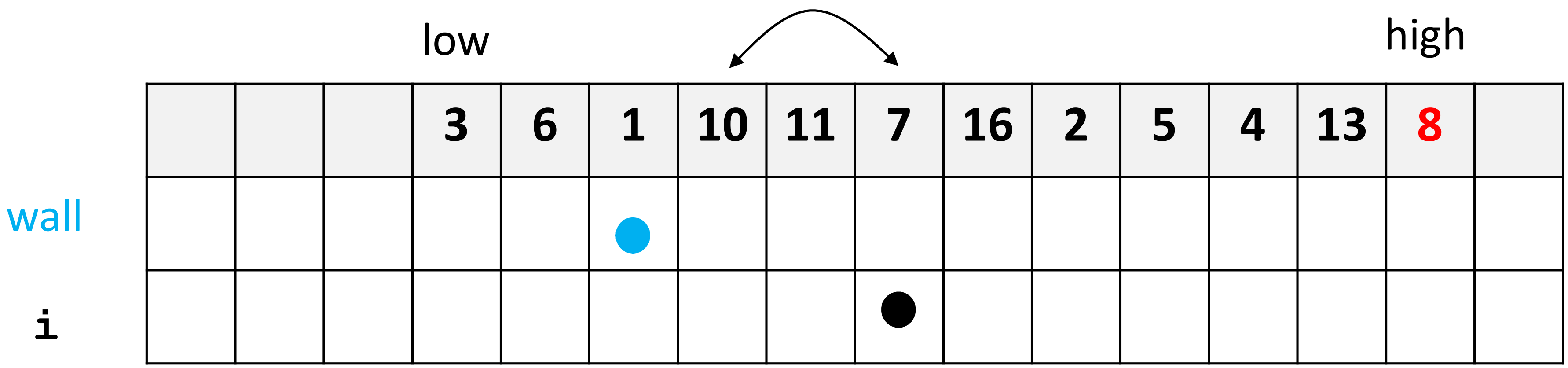

```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                 list.swap(wall, i)
      }
   return wall
}
```

1 <= 8 so we enter the block.
Increment wall and swap 11 and 1.



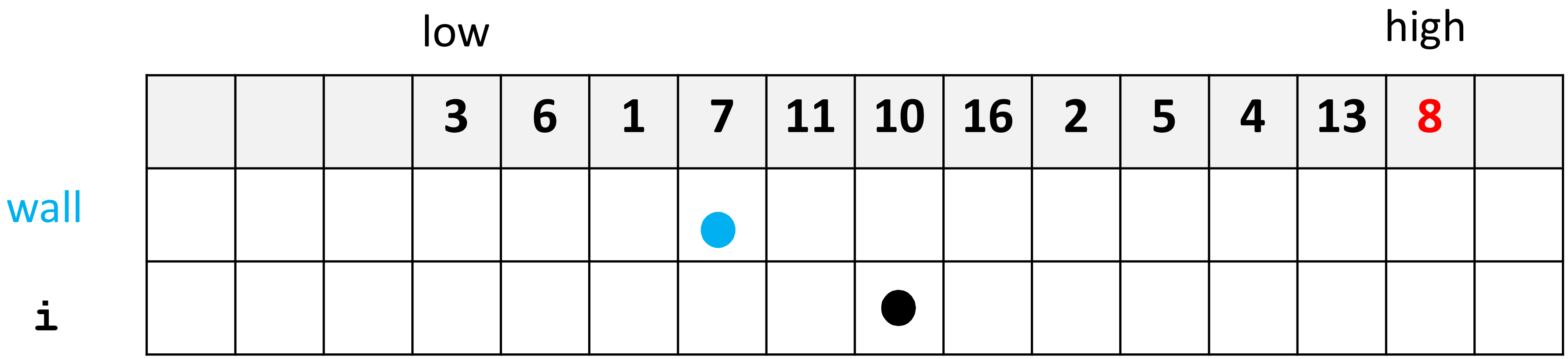

```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
         wall ++
         if  (wall != i)
                list.swap(wall, i)
      }
   return wall
}
```

At the end of the block, the situation is shown above.



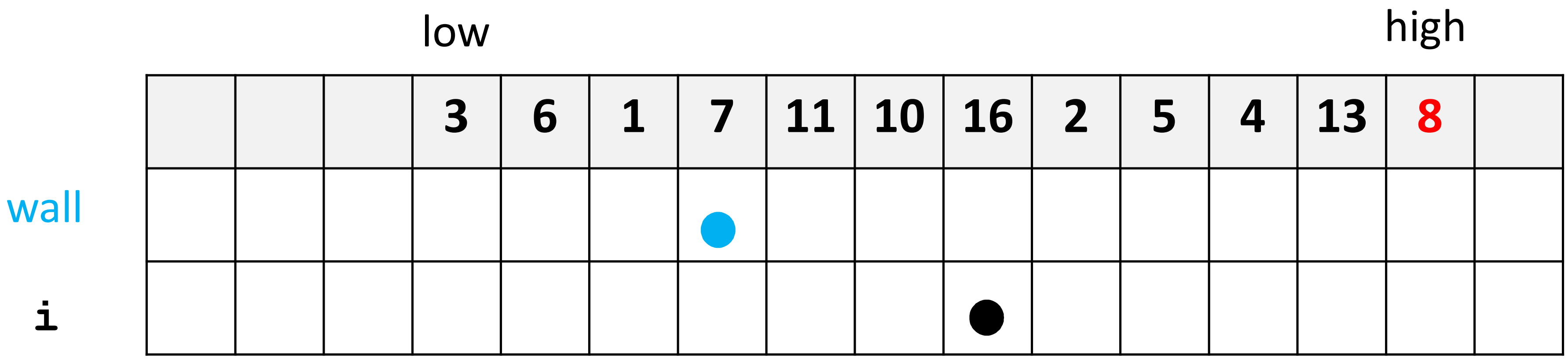

```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
        if ( list[i]  <=  pivot ){
           wall ++
           if  (wall != i)
                list.swap(wall, i)
        }
   return wall
}
```

7 <= 8 so we enter the block.
Increment wall and swap 10 and 7.



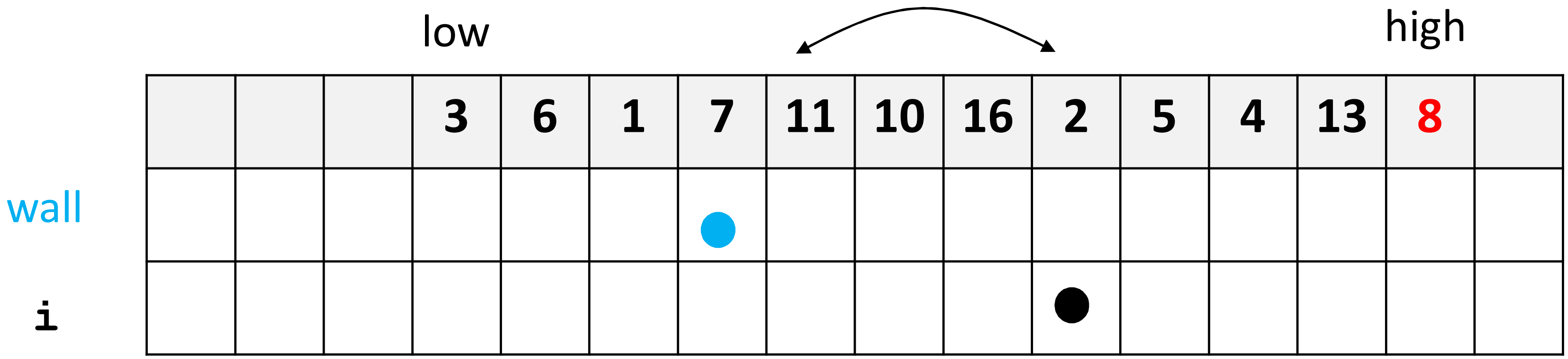

```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
        if ( list[i]  <=  pivot ){
             wall ++
             if  (wall != i)
                       list.swap(wall, i)
        }
    return wall
}
```

At the end of the block, the situation is shown above.




```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                list.swap(wall, i)
      }
   return wall
}
```

16 > 8 so we don’t enter the block



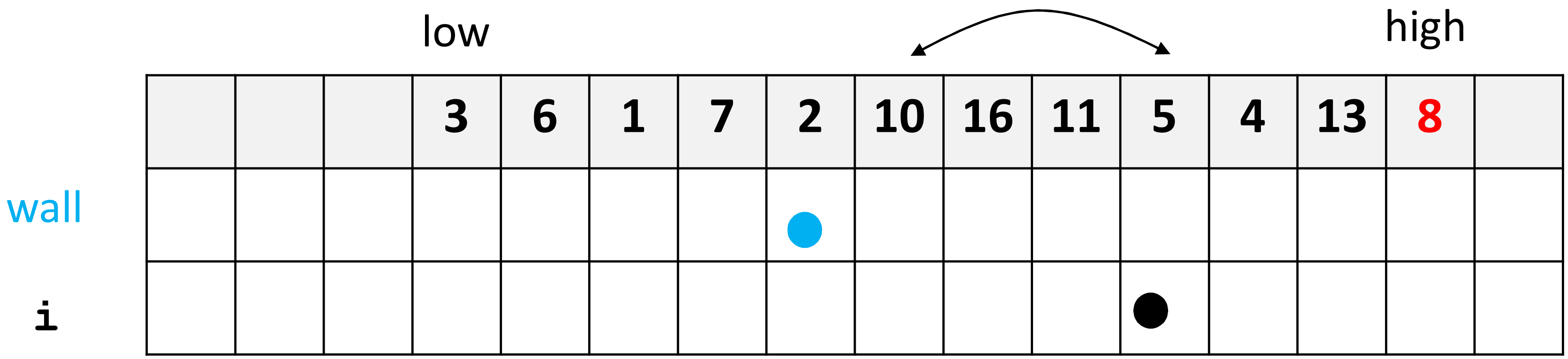

```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                list.swap(wall, i)
      }
   return wall
}
```

2 <= 8 so we enter the block.
Increment wall and swap 11 and 2.

				low											high	
				3	6	1	7	2	10	16	11	5	4	13	8	
wall								●								
i											●					

```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
        if ( list[i]  <=  pivot ){
             wall ++
             if  (wall != i)
                     list.swap(wall, i)
        }
    return wall
}
```

At the end of the block, the situation is shown above.



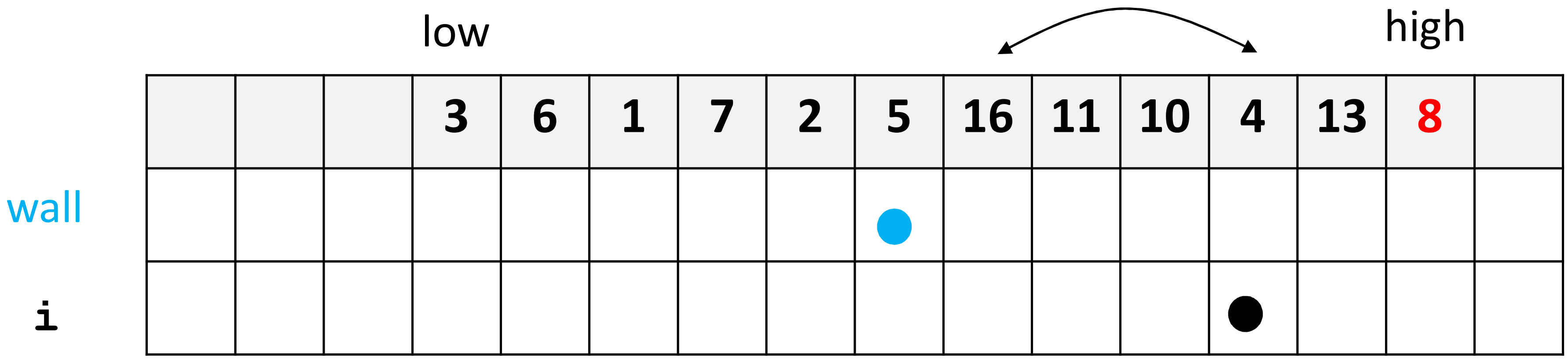

```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
       if ( list[i]  <=  pivot ){
           wall ++
           if  (wall != i)
                   list.swap(wall, i)
       }
    return wall
}
```

5 <= 8 so we enter the block.
Increment wall and swap 10 and 5.

			low										high	
			3	**6**	**1**	**7**	**2**	**5**	**16**	**11**	**10**	**4**	**13**	**8**	
wall									●						
i												●			

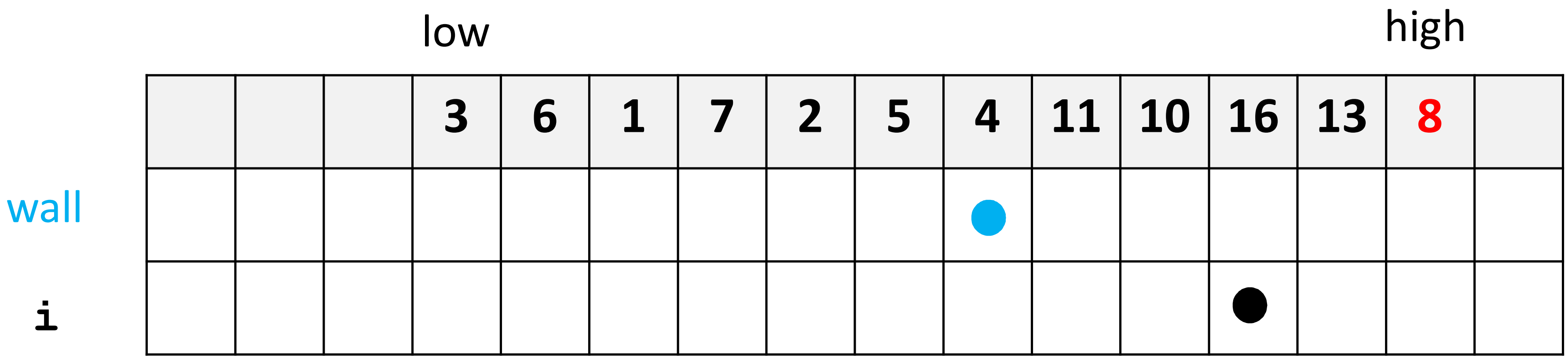
```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                  list.swap(wall, i)
      }
   return wall
}
```

At the end of the block, the situation is shown above.




```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
        if ( list[i]  <=  pivot ){
             wall ++
             if  (wall != i)
                     list.swap(wall, i)
        }
    return wall
}
```

4 <= 8 so we enter the block.
Increment wall and swap 16 and 4.



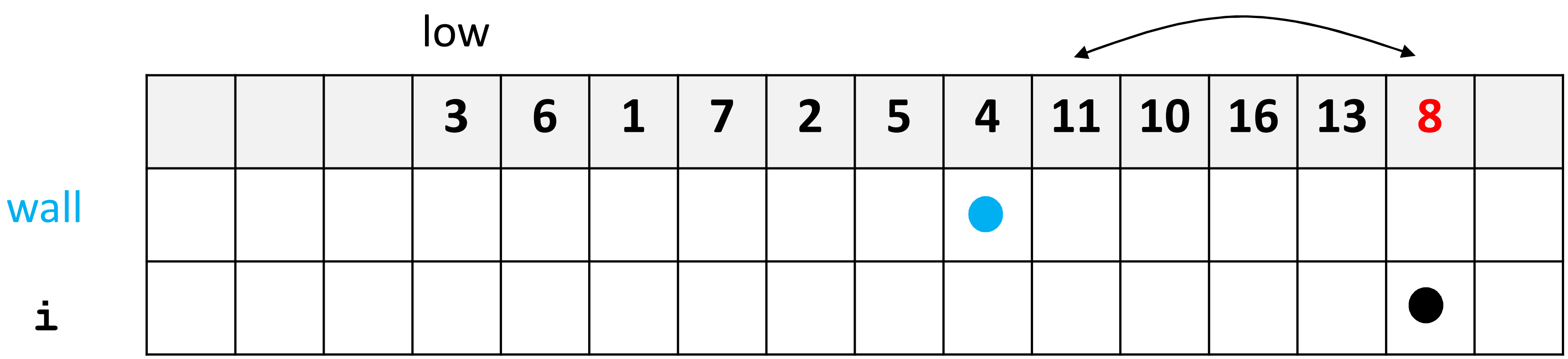

```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          if  (wall != i)
                list.swap(wall, i)
      }
   return wall
}
```

At the end of the block, the situation is shown above.

				low											high	
				3	**6**	**1**	**7**	**2**	**5**	**4**	**11**	**10**	**16**	**13**	**8**	
wall										●						
i														●		

```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
        if ( list[i]  <=  pivot ){
            wall ++
            if  (wall != i)
                   list.swap(wall, i)
        }
    return wall
}
```

13 > 8 so we don’t enter the block.
At the end of the block, the situation is shown above.




```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
        if ( list[i]  <=  pivot ){
             wall ++
             if  (wall != i)
                     list.swap(wall, i)
        }
    return wall
}
```

END GAME:
8 <= 8 so we enter the block (always happens)
Increment wall and swap 11 and 8.

				low												
				3	6	1	7	2	5	4	8	10	16	13	11	
wall											●					
i															●	

```
partition(list, low , high )
   pivot =  list[high]
   wall = low - 1
   for (i = low ; i <= high;   i++)
         if ( list[i]  <=  pivot ){
             wall ++
             if  (wall != i)
                   list.swap(wall, i)
         }
    return wall
}
```

The final situation is shown above.
The pivot is at the wall.

To be discussed later in course...

- Best case performance of quicksort ?

- Worst case performance of quicksort ?

- ASIDE: Other versions of quicksort?
  - different way to compute partition in place
  - different ways to choose the pivot